Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

How NOT to write kernel drivers

Arjan van de Ven
Red Hat, Inc.
arjanv@redhat.com, http://people.redhat.com/arjanv

Abstract

Quit a few tutorials, articles and books give an introduction on how to write Linux kernel drivers. Unfortionatly the things one should NOT do in Linux kernel code is is either only a minor appendix or, more commonly, completely absent. This paper tries to briefly touch the areas in which the most common and serious bugs and do-nots are encountered.

1 Introduction

Quit a few tutorials, articles and books give an introduction on how to write Linux kernel drivers. Unfortionatly the things one should NOT do in Linux kernel code is is either only a minor appendix or, more commonly, completely absent.

With the growing popularity of Linux in the last few years, more and more vendors are trying to create linux drivers, and quite often that is done by giving an engineer the windows driver code and the assignment to have a linux driver ready in 4 weeks.

In my job as Red Hat Linux kernel maintainer such drivers quite often end up in my inbox with the request to include it. Surprisingly quite often these drivers share the same bugs and "please don't do *that*" things.

This text and the corresponding talk is intended to show several such bugs and why they are bad. Quite a few will be of the "Oh but of course" caliber but that's usually the case with bugs in hindsight.

My hope is that explaining why certain things are wrong is enough to prevent such things from cluttering up my inbox too much.

All of the code examples in this paper are from real code, however they have been edited slightly to fit the layout and non-essential bits are removed for clarity.

2 Allocating memory

The Linux kernel has a diverse API for allocating memory, unlike operating systems such as Microsoft Windows and SCO Unixware. Linux uses this set of functions and flags one the one hand to be able to more aggressively optimize the VM algorithms, and on the other hand to provide safeguards against out-of-memory deadlocks.

Quite often drivers that are ported from other operating systems try to abstract the multitude of allocators and flags into one function. Not

only does this void the VM tuning optimisations, it also leads to subtle and hard to debug bugs.

2.1 GFP_KERNEL and GFP_ATOMIC

Most kernel tutorials describe that you shouldn't use GFP_KERNEL in interrupt context because it can schedule (which is correct), and as a result the following "abstraction" is found quite often:

```
static int uhci_submit_iso_urb(
urb_t *urb)
{
...
   tdm = kmalloc(some_size,
        in_interrupt()
      ? GFP_ATOMIC : GFP_KERNEL);
...
}
```

from usb-uhci.c in kernel 2.4.9

which appears to take this "interrupt context" requirement into account. However the entire cunning plan falls appart when spinlocks enter the picture: in_interrupt() might return false even though scheduling is not allowed. Since scheduling by kmalloc() is not the common case, this bug won't often show up in casual testing.

2.2 vmalloc

Most tutorials warn about using kmalloc for allocating big areas of memory, and it seems a lot of people also notice these limits in practice:

```
static inline void *
osi_malloc(unsigned int size)
{
   void *ptr;
   if (size > 2*PAGE_SIZE)
      return vmalloc(size);
   ptr = kmalloc(size, GFP_NOFS);
   if (ptr)
      return ptr;
   return vmalloc(size);
}
```

from OpenGFS 0.99.2

There are several problems with such an approach. The first one is performance: `vmalloc()`'ed memory requires more TLB's, which are a rare resource on most CPUs.

A more serious problem arises when the `osi_free()` routine gets involved: it calls `vfree()` for the `vmalloc()`ed memory, and it's illegal to call `vfree()` from interrupt context. This abstraction makes it okay to call the `osi_free()` function from interrupt context sometimes, but not for big allocations.

A third issue that most tutorials don't mention is that while you can use `vmalloc()` to allocate larger chunks of memory, the total amount you can allocate is rather limited, in the order of 64 Megabytes on modern machines (depending on what PCI hardware is present).

2.3 Other subtleties

Abstractions of the Linux memory allocators often also hides the deadlock avoidance mechanisms Linux provides. In the write path of a filesystem, using `GFP_KERNEL` or `GFP_ATOMIC` will lead to deadlocks eventually. Such deadlocks might not show up in your testing, but Murphy's law guarantees that your first big customer will hit the deadlock on December 24th at 6pm. `GFP_NOFS` is there

for a reason, as is GFP_NOIO for block device drivers. Using an abstraction for the allocator might appear to make your life easier by not having to understand these issues, but sooner rather than later it'll come back and haunt you—or, worse, your users.

3 Synchronisation primitives

The Linux kernel provides a reasonably complete set of synchronisation primitives in the form of semaphores and spinlocks (both in the normal form as the reader/writer variant). Rusty's hamster even wrote documentation about when to use what primivive. These primitives however do not form a perfect match with that Microsoft Windows or Unixware provide as primitives.

Not seldom do I find self-written synchronisation primitives in submitted drivers, and as a general rule they are all buggy.

```
/*
 * EnterCriticalSection:
 */
unsigned long
EnterCriticalSection(
DevInfo_pt pDev)
{
  int retval;
#ifdef __SMP__
  for (;;) {
    retval=test_and_set_bit(SEMA_SRL,
      &pDev->Sema);
    if (!retval)
       break;
    sleep_on(&pDev->WaitQ);
  }
  return(retval);
#else /* __SMP__ */
  if (pDev->Sema)
    return(-1);
  pDev->Sema=-1;
  return(0);
#endif
}
```

The code above is a typical example of driver-code that tries to emulate a primitive from another OS. The counterpart was as follows:

```
/*
 * LeaveCriticalSection
 */
void LeaveCriticalSection(DevInfo_pt
pDev)
{
#ifdef __SMP__
  clear_bit(SEMA_SRL,&pDevi->Sema);
  wake_up(&pDev->WaitQ);
#else
  pDev->Sema0;
#endif
}
```

It's tempting to leave the "what's wrong with that" as an excercise to the reader; however it's better to nip such code in the but so here goes a two-cpu example with 2 tasks, task A is holding the lock on cpu 1 and task B is trying to acquire the lock on cpu 2. Figure 1 shows how the mentioned code will leave task B sleeping without it ever getting the critical section.

	CPU 1	CPU 2	State
T0	....	....	Task A has has the sema bit set
T1	....	test_and_set	Task B tries and sees someone has the sema bit
T2	clear_bit	....	Task A releases with a clear_bit()
T3	wake_up	....	Task A waits up all waiters on the waitqueue
T4	....	sleep_on	Task B puts itself on the waitqueue and sleeps
T5	....	....	There's nobody left to wake task B ....

Figure 1: Timing diagram of the deadlock

Now it's very well possible to try to fix the above code to not have this deadlock. However, it's far simpler to just use the linux

semaphores which provide all the functionality required for this case.

Another problem with “home made locks” is that they do not work on architectures that reorder instructions and/or memory accesses aggressively, such as PowerPC or Power4.

One thing pops up rather often in drivers ported from other operating systems: recursive locks. The linux kernel provides no recursive locks (with the exception of the Big Kernel Lock, see section 4). The philosophy is that locking should be taken into account when designing your code, and in that case you don’t need recursive semaphores or spinlocks. Recursive locks have all sorts of subtle deadlock issues which are beyond the scope of this text and while you can write correct recursive locks, just say no.

4 SMP

Despite popular belief, SMP safety is not something you “weld” into your code as a hindsight. SMP safety is something you need to take into account right from the start. Makeing a (largish) piece of code SMP safe in hindsight leads to all kinds of lock-ordering nightmares, makes you wish there were recursive locks, and generally results in a suboptimal solution. While I could give numerous drivers as examples of how to not do it, the main kernel with the Big Kernel Lock (BKL) is the best example of this. The BKL was put in to make the kernel work on SMP, in hindsight—and well, it still results with nightmares with dozens and dozens of races. It has been taking 5 years so far to fix all the core subsystems to have proper locking of their own.

With the merging of the preemptible kernel patch in the 2.5 series of the kernel, SMP safety is even more important than before. With preemption turned on, your code can be interrupted at any point and acts as if it’s running on an SMP machine.

There’s a simple list of questions you should ask yourself for all code you write:

- What prevents the data I’m working on from beeing freed under me
- What prevents my module from being unloaded under me
- What prevents a user from opening/closing my device here
- What do I not want to happen to the data/device I’m working with right now...
- . . . and what makes sure that that doesn’t happen

Such a list can never be complete obviously, and the only weapon you can use to make your code SMP safe is your brain. Of course it helps if you have access to SMP hardware, and even booting an SMP kernel if you have only one CPU will allow you to find certain types of deadlocks.

5 All the world is not a VAX

```
#ifdef ALPHA
#define U32     unsigned int
#else
#define U32     unsigned long
#endif

from drivers/scsi/inia100.h,
kernel 2.4.9
```

The Linux kernel works on several architectures, not just Intel x86. The kernel API has evolved in a way that makes drivers basically automatically portable between architectures. That is, if the API is followed and no strange assumptions are made, as was done in the inia100.h example. Unfortunately, even if you don't follow the API the driver might at least *appear* to work on a normal PC, since the PC architecture makes certain consistency guarantees and has certain behavior that other platforms can't provide.

5.1 PCI posting

One of the most common, and hardest to debug, problems in PCI device drivers is the lack of dealing with *PCI posting*. *PCI posting* is the effect where the cpu writes some data to a certain PCI device, the PCI bridge (or, rather, any component between the CPU and the actual device) is allowed to buffer this write as long as it wants. The only constraint to this buffering is that a read operation from a device will not complete before all pending writes to this device are completed, hence effectively forcing a "flush" of all pending writes.

A typical example of how this can easily go wrong is shown below:

```
//————————————————————————-
// Procedure:    eeprom_stand_by
//
// Description: This routine lowers the
//      EEPROM chip select (EECS) for a
//      few microseconds.
//————————————————————————-
static void
eeprom_stand_by(struct e100_priv
*adapter)
{
   u16 x;
   x = readw(&CSR_EEPROM(adapter));
   x &= ~(EECS | EESK);
   writew(x, &CSR_EEPROM(adapter));
   udelay(EEPROM_STALL_TIME);
   x |= EECS;
   writew(x, &CSR_EEPROM(adapter));
   udelay(EEPROM_STALL_TIME);
}
```

`from the Intel e100 driver in kernel 2.5.6`

The intent of the programmer clearly is to clear some bit in the cards memory space, wait a certain amount of time, enable it again and wait some more time, effectively creating a periodic waveform if the routine is called in sequence. However, due to posting, the first writew() might not reach the card for a long time and the udelay() is therefore totally missing the intended goal, it just warms the cpu a bit. The end result in this case is that the eeprom contents is written out incorrectly to the card.

The fix is obvious (and present in later kernels); just adding a `readl()` immediately after both `writew()` to read and discard some data from the card is enough.

Only recently started the more advanced PC chipsets to do more aggressive posting, so this bug probably will not, or only seldom, show up on home PC's. IA64 and other architectures have had more aggressive posting in the chipsets for a longer time already.

5.2 GFP_DMA

The GFP_DMA memory allocation flag was originally intended to allocate ISA bus DMA-able memory, however several architectures

give a different meaning to it nowadays. Since the demise of the ISA bus, GFP_DMA should not be used in drivers *at all*; the PCI DMA API has well defined ways of allocating and mapping memory for PCI use. The example below shows the Intel e100 driver abuse GFP_DMA to work around a performance issue in early IA64 architecture code that deals with a missing IOMMU chip.

```
#if (defined __ia64__)
   new_skb =
      __dev_alloc_skb(skb_size,
      GFP_ATOMIC|GFP_DMA);
   // Try to alloc non-DMA skb if
   // failed to get from the DMA zone

   if (new_skb==NULL){
      new_skb =
        dev_alloc_skb(skb_size);
   }
#else
   new_skb =
        dev_alloc_skb(skb_size);
#endif
```

`Intel e100 1.8.38` While the code will work correctly for current IA64 systems, it's not guaranteed that newer IA64 machines won't have an IOMMU and that GFP_DMA doesn't actually result in PCI-reachable memory. The obviously right thing here was to fix the performance bug in the architecture code, or even making the architecture use the highmem mechanism.

5.3 Assuming PC resources

```
static void
qla2100_putc(int8_t c)
{
...
   /* BAUD rate divisor LSB. */
   OUTB(0x3f8+3, 0x83);
   /* BAUD rate divisor MSB. */
   /* 0xC = 9600 baud */
   OUTB(0x3f8, 0xc);
...
}
```

`Qlogic 2x00 v5.31 fiber channel driver` The above example is taken from a fibre channel driver. The purpose is to have some sort of serial console for tracing and debugging (let's pretend for a minute that Linux doesn't already have generic serial console code).

Hardcoding architecture constants like `0x3f8` (even if they haven't changed in the last 20 years) is just inviting trouble. Some day someone will want to run your code on an IA64, or a MIPS or even an ARM machine, and all hell breaks loose. In addition, these "constants" actually recently started changing with the advent of the legacy-free PC's.

Another serious issue is that this code uses IO resources without registering them with the kernel; one can assume that even the most expensive intelligent UPS gets confused when a driver like this interferes with the monitoring application which happens to be connected to this serial port (and yes, sysadmins do get grumpy when in the middle of the night a bad sector results the driver turning off the UPS due to error handling calling this debug code).

5.4 On-disk and wire formats

Two of the major differences between architectures that are actually visible to kernel drivers are byte order and structure padding.

Both these items you normally don't have to care about; however, they do become important

when you put information on persistent storage (when writing a filesystem for example) or when sending information over a network.

The JFFS2 filesystem was sloppy in this respect and stored its metadata on the (flash) device in CPU byte order. After the filesystem was in use for several months, on several architectures, people started complaining that they couldn't take their device from one system (powerpc) to another (x86) while still being able to read the data. David Woodhouse can testify that making a filesytem auto-sensing, bi-endian is not something you want to do.

Using a defined byte order for the on disk format is certainly preferable. Verifying the correctness of code in this respect is normally non-trivial if you only have access to x86 machines; EXT3 uses the cunning trick to specify all on-disk data (in the journal) in Big Endian order so that any missing byte-order correction will be noticed immediatly on a PC.

Structure padding is a similar issue:

```
struct example {
    char foo;
    u64  bar;
};
```

in the example above the size of `struct example` will depend on the architecture. Different architectures have different requirements for minimal alignment of data and the compiler will add invisible padding between `foo` and `bar` to satisfy the architecture requirements.

This has 3 important consequences when this structure needs to be written to persistent storage or the network:

1. The size of the structure differs per architecture
2. The location of the `bar` data is in a different place in the bytes that make up `struct example`
3. The code below is *not* enough to clear the entire structure

```
struct example *blah;
...
blah = kmalloc(sizeof(*blah),
       GFP_KERNEL);
if (!blah)
    return -ENOMEM;
blah->foo = 3;
blah->bar = 40;
...
```

The code above does not clear the (invisible) padding, and for in-kernel use that's not a problem. However when exposing this struct outside the kernel, it is very important to realize that the padding bytes are uninitialized and hence can contain just about anything that ever was in memory, including the root password or parts of gpg keys. Things like that usually make sure that your module ends up on bugtraq just before the meeting with a big potential customer.

6 Using `int` for flags

A common bug that causes portability issues is the use of an `int` variable for storing the CPU flags in with `spin_lock_irqsave`.

```
void
mgsl_sppp_tx_timeout(
struct net_device *dev)
{
   struct mgsl_struct *info =
```

```
      dev->priv;
   int flags;
...
   spin_lock_irqsave(
        &info->irq_spinlock, flags);
   usc_stop_transmitter(info);
   spin_unlock_irqrestore(
        &info->irq_spinlock, flags);
}
```

from drivers/char/synclink.c
kernel 2.4.9

On 64 bit architectures, the CPU flags are 64 bit, and the code such as quoted above will fail to work quite spectacularly. `spin_lock_irqsave` is specified to take an `unsigned long` variable for flags, and thankfully kernels 2.5.10 and later will cause a compiler warning if this isn't the case.

7 Files

Just about all kernel tutorials and books warn you to not open configuration files from inside the kernel. Unfortionatly that doesn't seem to stop people from doing so anyway.

```
static void chandev_read_conf(void)
{
#define CHANDEV_FILE \
        "/etc/chandev.conf"
...
    set_fs(KERNEL_DS);
    if(stat(CHANDEV_FILE,
       &statbuf)==0)
    {
       set_fs(KERNEL_DS);
       if((fd=open(CHANDEV_FILE,
              O_RDONLY,0))!=-1)
       {
          curr=0;
          left=statbuf.st_size;
          while((len=read(fd,
               &buff[curr],left))>0)
          {
             curr+=len;
             left-=len;
          }
....
}
```

drivers/char/s390/misc/chandev.c
in kernel 2.4.9/s390

Apart from the aesthetic issues involving the kernel setting policy on userspace, and the problem that writing a secure parser is non-trivial (both of which doesn't seem to impress people enough to not write code like this), there is the issue of *which* file is it.

There are several reasons why this isn't as clear as it might look on first sigh. In recent 2.4 and 2.5 kernels, each process has its own namespace (and hence effectively its own root directory). Autoloading the driver will result in the module using `init`'s namespace, while manually loading it by root will result in the module using the namespace root currently happens to be in. `chroot` will have similar surprise effects, as will loading the module from an initial ramdisk.

The initial ramdisk case just makes a system harder to administer, but the other two cases actually carry a slight security risk: while one needs to be root to load modules, root might not realize he is not in the `init` namespace; this gives a non-root user the ability to feed a config file to the kernel (and either merely affect the settings, or worse, exploit vulnerabilities in the parser).

8 Just yuck

8.1 Overriding system calls

There are certain things that should just not be done in kernel space. The most evil one is overriding or adding system calls from modules. For historical reasons, the `sys_call_table` symbol is exported for the use in the Linux ABI patches that allow the kernel to run binaries from other linux-like operating systems (Unixware, Solaris etc). Linux ABI only needed this to call existing function calls, and has long since been fixed to do so directly.

Unfortionatly this export seems to have opened the door for other modules to override existing system calls with different behavior. The mix of modules that do this consists of several filesystems of IBM origin, binary only security tools, and oprofile, a performance monitoring tool.

```
void oplock_init(void)
{
  TRACE0(TRACE_SMB, 5,
    TRCID_OPLOCK_INIT,
    "Loading kernel"
    " oplock support\n");
  old_sys_fcntl =
     sys_call_table[__NR_fcntl];
  old_sys_fcntl64 =
     sys_call_table[__NR_fcntl64];
  sys_call_table[__NR_fcntl] =
     (long)&newsys_fcntl;
  sys_call_table[__NR_fcntl64] =
     (long)&newsys_fcntl64;
}
```

glue layer for IBM GFPS binary only (C++) filesystem

Apart from the SMP races in such code (it's impossible to get the locking against module unloading working), overriding system calls is just too evil for words.

8.2 Depending on userspace program names

Writing kernel code that depends on filenames of programs in userspace prevents people from arranging their system the way they want, and also can lead to "interesting" surprises.

Sometimes filenames are unavoidable; the kernel has a series of filenames for `init` it tries when booting, while allowing a command line override. `modprobe` and `/sbin/hotplug` are also called by the kernel to provide improved plug-and-play behavior; however the filenames of both are /proc settings, and the bootup scripts are expected to set the proper values if the provided defaults are not correct.

A different kind of problem is if the kernel decides to behave differently based on the name of the program that is running:

```
Boolean
cxiIsSambaThread()
{
   return (!strcmp(current->comm,
      "smbd"));
}
```

glue layer for IBM GFPS binary only (C++) filesystem

There is no guarantee that Samba is the only binary that is named *smbd*, and changing behavior based on such a test can lead to interesting surprises for other programs. There even can be a security angle if this test is used for enhancing permissions.

8.3 Long udelay

I have been told that drivers for Microsoft Windows can sleep in a lot more places than Linux drivers. Several drivers I've seen that are ported from Windows or are shared with Windows depended on this and the Linux variant of the driver gets into trouble as a result. The common solution between such drivers seems to be to just use the busy-waiting `udelay()` macro.

```
static uint8_t
qla2100_mailbox_command(....)
{
...
  cnt = 0x100000*2;    /* 22 secs */
  for( ; cnt > 0 && !ha->flags;
    cnt-- )
  {
    /* Check for pending interrupts. */
     data = RD_REG_WORD(
       &reg->istatus);
     ...
     udelay(10);
  }
}
```

`from Qlogic qla2x00 driver version 4.28`

In the above driver code snippet the `qla2100_mailbox_command()` function, which has to be called with the `io_request_lock` spinlock acquired and local interrupts disabled, busy waits for about 20 seconds. This is thankfully not the common case, but since the spinlock in question is required for doing any IO at all, 22 seconds is usually enough to kill a system (especially if hardware watchdogs are involved). And even if it doesn't, writing a driver like this will not make you popular with the low latency people.

There is no clear-cut way to fix such problems in an afternoon. In the qla2x00 case, it required a complete redesign of all the locking and some surgery in the structure of the driver. As a side effect the lock hold times and contention reduced significantly (even ignoring the long busy waits since those are not really in fast paths); fixing these issues pays off in more than one way.

8.4 Floating point

Everybody knows using floating point in the kernel isn't allowed (with the exception of carefully selected places where MMX is used to speed up RAID XOR performance and such). However it's rather easy to put floating point in by accident:

```
#define NTSC 14.31818
#define calc_freq(n,m,k) \
   ((NTSC * (n+8))/((m+2)*(1<<k)))
...
int fi;
fi = calc_freq(n,m,k);
```

`drivers/video/trident_fb.{c,h} 2.4.18pre`

The first define is actually in the header file and all the use is in the C file, where the author probably no longer realized the NTSC constant was a float. Compiling the kernel with the `-msoft-float` flag finds such problems fast before random userspace floating point results get corrupted.

9 Conclusion

This paper only touches the tip of the iceberg of problems found in kernel modules; however I

hope that it has become clear that adding abstraction layers for the Linux kernel API to glue drivers from other operating systems to Linux, is a bad idea. Beyond that it's mostly a matter of common sense and having a good design of a driver; due to the open source nature of the Linux kernel there's plenty of good (but also of bad) examples available for borrowing a design idea.

10 Legalese